Search-based testing of procedural programs: iterative single-target or multi-target approach?

Scalabrino, Simone ; Grano, Giovanni ; Di Nucci, Dario ; Oliveto, Rocco ; De Lucia, Andrea

Abstract: In the context of testing of Object-Oriented (OO) software systems, researchers have recently proposed search based approaches to automatically generate whole test suites by considering simultaneously all targets (e.g., branches) defined by the coverage criterion (multi-target approach). The goal of whole suite approaches is to overcome the problem of wasting search budget that iterative single-target approaches (which iteratively generate test cases for each target) can encounter in case of infeasible targets. However, whole suite approaches have not been implemented and experimented in the context of procedural programs. In this paper we present OCELOT (Optimal Coverage sEarch-based tooL for sOftware Testing), a test data generation tool for C programs which implements both a state-of-the-art whole suite approach and an iterative single-target approach designed for a parsimonious use of the search budget. We also present an empirical study conducted on 35 open-source C programs to compare the two approaches implemented in OCELOT. The results indicate that the iterative single-target approach provides a higher efficiency while achieving the same or an even higher level of coverage than the whole suite approach.

DOI: https://doi.org/10.1007/978-3-319-47106-8_5

Originally published at:
Scalabrino, Simone; Grano, Giovanni; Di Nucci, Dario; Oliveto, Rocco; De Lucia, Andrea (2016). Search-based testing of procedural programs: iterative single-target or multi-target approach? In: Sarro, Federica; Deb, Kalyanmoy. Search Based Software Engineering : 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings. Cham: Springer, 64-79.
DOI: https://doi.org/10.1007/978-3-319-47106-8_5

Search-based Testing of Procedural Programs: Iterative Single-Target or Multi-Target Approach?

Simone Scalabrino[1], Giovanni Grano[2], Dario Di Nucci[2], Rocco Oliveto[1], and Andrea De Lucia[2]

[1] University of Molise, Italy
simone.scalabrino@unimol.it, rocco.oliveto@unimol.it
[2] University of Salerno, Italy
g.grano1@studenti.unisa.it, ddinucci@unisa.it, adelucia@unisa.it

Abstract. In the context of testing of Object-Oriented (OO) software systems, researchers have recently proposed search based approaches to automatically generate whole test suites by considering simultaneously all targets (*e.g.,* branches) defined by the coverage criterion (*multi-target approach*). The goal of whole suite approaches is to overcome the problem of wasting search budget that iterative single-target approaches (which iteratively generate test cases for each target) can encounter in case of infeasible targets. However, whole suite approaches have not been implemented and experimented in the context of procedural programs. In this paper we present OCELOT (Optimal Coverage sEarch-based tooL for sOftware Testing), a test data generation tool for C programs which implements both a state-of-the-art whole suite approach and an iterative single-target approach designed for a parsimonious use of the search budget. We also present an empirical study conducted on 35 open-source C programs to compare the two approaches implemented in OCELOT. The results indicate that the iterative single-target approach provides a higher efficiency while achieving the same or an even higher level of coverage than the whole suite approach.

Keywords: Test Data Generation, Search-based Software Testing, Genetic Algorithm

1 Introduction

Software testing is widely recognized as an essential part of any software development process, representing however an extremely expensive activity. The overall cost of testing has been estimated at being at least half of the entire development cost, if not more [5]. Generating good test cases represents probably the most expensive activity in the entire testing process. Hence, testing automation is receiving more and more attention by researchers and practitioners in order to increment the system reliability and to reduce testing costs. In this context, search-based algorithms have been efficiently used for the test data generation problem [24]. Specifically, such approaches can be used to generate test data with

respect to a coverage criterion (typically, branch coverage) aiming at covering a specific target at a time (typically, a branch). In order to obtain a complete test suite, the approach is executed multiple times, changing the target branch each time, until all branches are covered or the total search budget, *e.g.,* time available, is consumed (*iterative single-target test suite generation*).

The iterative single-target test suite generation has two important limitations [13]. First, in the program under test there might be branches that are more difficult to cover as compared to others or there might be infeasible branches. Thus, the search algorithm may be trapped on these branches wasting a significant amount of the search budget [13]. Second, the order in which target branches are selected can have a large impact on the final performance. In order to mitigate these limitations, Fraser and Arcuri [13] proposed the *whole test suite* approach, where instead of searching iteratively for tests that cover specific branches, the search algorithm searches for a set of tests (test suite) that covers all the branches at the same time. Following the same underlying idea, Panichella *et al.* [29] recently proposed MOSA (Many-Objective Sorting Algorithm), an algorithm where the *whole test suite* approach is re-formulated as a many-objective problem, where different branches are considered as different objectives to be optimized. MOSA is able to achieve higher coverage or a faster convergence at the same coverage level as compared to a single-objective whole test suite approach [29]. Nevertheless, whole suite approaches have been introduced in the context of Object-Oriented (OO) software systems and they have never been experimented and compared to iterative single-target approaches in the context of procedural programs.

In this paper we present a new test data generation tool for C programs named OCELOT (Optimal Coverage sEarch-based tooL for sOftware Testing) which implements both the many-objective whole suite approach MOSA [29] and a new iterative single-target approach named LIPS (Linearly Independent Path based Search) designed to efficiently use the search budget and re-use profitable information from previous iterations. We also conduct an empirical study on 35 open-source C programs to compare the two test data generation approaches. The results achieved indicate that, if targets are selected aiming at parsimoniously using the search budget, the iterative single target method provides comparable or better performance than the more sophisticated whole suite approach.

The remainder of this paper is organized as follows. Section II summarizes background information and presents the related literature. Section III presents OCELOT and the implemented test data generation approaches (MOSA and LIPS). The results of the empirical study are reported in Section IV, while Section V concludes the paper highlighting future research directions.

2 Background and Related Work

Search-based software testing approaches apply search-based algorithms—such as Hill Climbing [16], Simulated Annealing [32], Alternating Variable Method

(AVM) [19] and Genetic Algorithm (GA) [36]—to automatically generate test input data.

The design of any search algorithm for a specific optimization problem usually requires the definition of the solution representation and the fitness function. In the context of test data generation, a solution is represented by a set of test inputs [24]. The fitness function, instead, highly depends on the coverage criterion. Usually, branch coverage is used as code coverage criterion [18] [26] [30] [32] [36] [35]. Specifically, the fitness function is mainly based on two measures: *approach level* [30] and *branch distance* [18]. The *approach level* represents how far is the execution path of a given test case from covering the target branch, while the *branch distance* represents how far is the input data from changing the boolean value of the condition of the decision node nearest to the target branch. As the branch distance value could be arbitrarily greater than the approach level, it is common to normalize the value of the branch distance [1] [35].

The first search-based approaches for test data generation defined in the literature select the branches to covered incrementally (single-target strategy) [35]. A simple single-target strategy for branch coverage could be summarized as: (i) enumerate all targets (branches); (ii) perform a single-objective search, for each target, until all targets are covered or the total search budget is consumed; (iii) combine all generated test cases in a single test suite. Among the many tools, prototype tools and framework that implemented the early single-target approaches, we can mention *TESTGEN* [10], *QUEST* [6], *ADTEST* [14] and *GADGET* [27]. A typical problem of tools that generate test cases for programs developed in C is the handling of pointers. Lakhotia *et al.* [20] try to solve this problem introducing a new approach, named AVM+. Such an approach is implemented in *AUSTIN*, an open-source tool for automated test data generation in C [20].

It is worth noting that the generated test cases need to be manually refined to specify for each of them the oracle [4]. This means that the higher the number of generated test cases the higher the effort for the tester to generate the oracle [4]. Such a problem has recalled the need to consider the *oracle effort* when generating the test suite. A simple solution for solving this issue consists of reducing the size of the generated test suite. With this goal, Oster and Saglietti [28] introduced a technique, based on control and data flow graph criteria, aimed at maximizing the code coverage and minimize the number of test cases. Afterwards, Harman *et al.* [15] proposed three formulations of the test case generation problem aiming at reducing oracle effort: (i) the *Memory-Based Test Data Reduction* that maintains a set of not yet covered target branches during the iterations; (ii) a greedy set cover algorithm; and (iii) a *CDG-Based* algorithms. In the third formulation the fitness function is split in two parts: the first consisting in the sum of approach level and branch distance and the second considering the *collateral coverage* (serendipitously achieved). All such formulations were implemented in IGUANA [25], a tool designed to simplify the implementation of different single-target approaches and the comparison among

them. Finally, Ferrer *et al.* [11] dealt with coverage and oracle cost as equally important targets.

Besides the aforementioned improvements, single target approaches still suffer of two important limitations: (i) they can waste a significant amount of the search budget trying to cover difficult or infeasible branches; (ii) the search for each target is typically independent, and potentially useful information is not shared between individual searches. In order to mitigate such problems, Fraser and Arcuri [13] proposed the *whole test suite generation* approach, implemented in the Evosuite tool [12]. This approach evolves testing goals simultaneously. A candidate solution is represented as a test suite and the fitness function is represented by the sum of all branch distances and approach levels of all the branches of the program under test. An experimentation conducted on 1,741 Java classes showed that the whole suite approach achieves higher coverage than single target approaches (on average 83% vs 76%) and produces smaller test suites in 62% of the cases. Nonetheless the whole suite approach proposed by Fraser and Arcuri [13] has a drawback: it tends to reward the whole coverage more than the coverage of single branches [29]. Thus, in some cases, trivial branches are preferred to branches that are harder to cover, affecting the overall coverage. To mitigate such a problem, Panichella *et al.* [29] formulate the test data generation problem as a many-objective problem. In particular, the authors consider the branch distance and the approach level of each branch as a specific fitness function. In this reformulation, a test case is considered as a candidate solution, while fitness is evaluated according to all branches at the same time. Since the number of fitness functions could be very high, the authors introduced a novel many-objective GA, named *MOSA* (Many-Objective Sorting Algorithm), and integrated the new approach in Evosuite. The results of an empirical evaluation conducted on 64 Java classes indicated that MOSA produces better results compared to a single-objective whole test suite approach, *i.e.,* MOSA achieved a higher coverage or a faster convergence when the coverage level is comparable.

From the analysis of the state-of-the-art—to the best of our knowledge—emerges that whole test suite approaches have been never experimented and compared to single target approaches in the context of procedural programs. Moreover, none of the tools presented in this section implements both single-target and multiple-target approaches for procedural programs. In this paper we bridge this gap by introducing a new tool for search-based test data generation for C programs. The tool implements both a whole test suite approach and a novel iterative single-target approach, allowing to compare, for the first time, single-target and a multiple-target test data generation approaches in the context of procedural programs.

3 OCELOT in a Nutshell

OCELOT (**O**ptimal **C**overage s**E**arch-based too**L** for s**O**ftware **T**esting) is a new test suite generation for C programs implemented in Java. Unlike previous tools for C programs, OCELOT automatically detects the input types of a given C

function without requiring any specification of parameters. In addition, the tool handles the different data types of C, including structs and pointers and it is able to produce test suites based on the Check unit testing framework[3]. As well as all the tools presented in Section 2, OCELOT is not able to generate oracles: such a task is delegated to a human expert.

OCELOT includes two different target selection strategies. The first strategy is represented by MOSA [29], while the second one is represented by LIPS (Linearly Independent Path based Search), a technique inspired by the baseline method proposed by McCabe *et al.* [34] and never used for search-based test data generation before. We define LIPS in the context of this study and we do not use a state-of-the-art technique in order to have a fair comparison between the two families of approaches, *i.e.,* iterative single-target and multi-target. Indeed, LIPS was properly customized to share with whole suite approaches the main goals of efficient use of the search budget and re-use of profitable information from previous iterations.

The proposed iterative single-target approach is independent of the search algorithm used to generate test data. However, we decided to use GA to have a fair comparison with MOSA that is based on a many-objective GA, using *JMetal* [9], a Java-based framework for multi-objective optimization with metaheuristics. We used the default GA configuration parameters, the *SBX-Crossover* for the crossing-over, the *Polynomial Mutation* for the mutation, and the *Binary Tournament* operator for selecting the fittest individuals. The GA configuration and the genetic operators are exactly the same for both the whole suite and the iterative single-target approach. It is worth noting that a solution in OCELOT is represented as a list of input data [19], differently from Evosuite [12]. Therefore, the version of MOSA implemented in OCELOT differs from the original one as for this aspect. In the following we provide more details on the two approaches.

Many-Objective Sorting Algorithm (MOSA). MOSA reformulates the test suite generation problem as a many-objective optimization problem [29]. A solution is a test case and each objective represents how far a test case is from covering a specific branch.

As first step MOSA randomly generates an initial set of test cases. Such test cases represent the starting population of the genetic algorithm. In the generic i^{th} iteration (*generation*) of the genetic algorithm, *offspring* solutions are created from the actual population and added to a set R_t, together with the population P_i. All such solutions are sorted in Pareto-fronts $\mathbb{F}$, each of which has a specific *rank*. If a solution belongs to a Pareto-front with rank a, it means that such a solution is better than all the solutions which belong to a Pareto-front with rank $b > a$. MOSA generates the population for the next generation P_{i+1} starting from the Pareto-front with rank 0, and adding whole fronts until a $\mathbb{F}_d$, so that the addition of such a front would make the population larger than maximum size, specified through the parameter PS. Anyhow, it may be necessary to add some of the solutions belonging to $\mathbb{F}_d$ to the next population P_{i+1} in order to

[3] https://libcheck.github.io/check/

reach the maximum population size. MOSA promotes diversity adding to P_i solutions from $\mathbb{F}_d$ that increase most the crowding distance.

In MOSA, the preference-sorting algorithm of Pareto-fronts has a key role. The main problem is that multi-objective algorithms, like Non-dominated Sorting Genetic Algorithm II (NSGA-II)[8], Strength Pareto Evolutionary Algorithm (SPEA2) [39] or Indicator Based Evolutionary Algorithm (IBEA) [38] do not scale efficiently and effectively for problems with more than 15 objectives (even less, in some cases) [22]. In the context of test suite generation, a program could have hundreds of branches. For this reason, MOSA introduces a novel sorting algorithm which is specific for the test case generation problem. $\mathbb{F}_0$ will contain the solutions that minimize the objective function relative to branches not covered yet. Such an expedient allows to include solutions that could lead to a strong improvement of the coverage. The preference-sorting algorithm ranks other solutions using the non-dominated sorting algorithm used by NSGA-II [8]. Such an algorithm focuses only on objectives relative to uncovered branches, in order to concentrate the search in interesting areas of the search space. Test cases that cover specific branches are progressively saved in a separate data-structure: the *archive*. In each iteration of the genetic algorithm, the archive will be updated, so that if a solution is able to cover a previously uncovered branch, it is stored into the archive. At the end of the algorithm, the archive will contain the final test suite.

Whole test suite approaches, in general, and MOSA, in particular, have been designed to work on OO languages. Since in such a context unit testing is generally focused on classes, a test case is represented as a sequence of statements in order to handle many aspects such as instantiation, method calls and so on [31]. Conversely, in the context of procedural languages, a function can be considered as the unit to test. Thus, we properly customize MOSA in order to represent a test case as the input data (test data) of the function that has to be tested [24].

Linearly Independent Path based Search(LIPS). LIPS is an iterative single-target approach we designed with the goal of mitigating the main limitations of previous single-target approaches.

The target selection strategy exploited by LIPS takes inspiration from the baseline method proposed by McCabe *et al.* [34], which computes a maximal set of linearly independent paths of a program (a basis) [23]. This algorithm incrementally computes a basis, by adding at each step a path traversing an uncovered branch [34]. This means that executing all the paths in a basis implies the coverage of all branches in the control flow graph [23]. Similarly, LIPS incrementally builds a set of linearly independent paths by generating at each iteration a test case (and then a path) able to cover a still uncovered branch. It is worth noting that LISP does not need to generate test data for all linearly independent paths of a basis in case the maximal coverage is achieved in advance (due to collateral coverage).

The algorithm is partly inspired by Dynamic Symbolic Execution [21]. The first step is to randomly generate the first test case (t_0). For each decision node in the execution path of t_0 the uncovered branch of the decision is added to a

worklist. A random population which includes t_0 is then generated to be used by the second iteration of the algorithm. At the generic iteration i, the last branch added to the worklist is removed and used as a target of the search algorithm. If the search algorithm is able to find a test case that covers the target, a new test case, t_i is added to the test suite and all the uncovered branches of decision nodes on the path covered by t_i are added to the worklist. This procedure is iterated until the worklist is empty (i.e. all the branches are covered) or until the search budget, measured as number of fitness evaluations, is entirely consumed. Note that at each iteration the last branch added to the worklist is used as target of the search algorithm and the final population of the previous iteration is reused (seeding), since it likely includes the test case covering the alternative branch. In this way, we expect that the search algorithm will take less time to generate a test case able to cover the target branch.

Sometimes, a test case can cover some branches that are already in the worklist (*collateral coverage*). These branches are removed from the worklist and marked as "covered". On the other hand, it could happen that, while searching for the test case which covers a certain branch, some of the partial solutions generated by the search algorithm are able to cover other branches in the worklist. Such test cases are added to the test suite and the covered branches are removed from the worklist. It is worth noting that while this approach improves the search efficiency (time) and effectiveness (coverage), it might result in adding redundancy to the test suite. This issue will be discussed in Section 4.2.

Handling the budget in single-target approaches can be tricky. Allocating the remaining budget to the search for covering a specific branch could be very damaging, because budget will be wasted in case the target branch is infeasible or difficult to cover. An alternative budget handling policy consists of distributing equally the budget over the branches. In other words, if the total budget is SB (*e.g.,* number of fitness function evaluation) and the program contains n branches, a budget of $\frac{SB}{n}$ will be available for such branch. LIPS uses a *dynamic* allocation of the search budget. Specifically, at the iteration i of the test generation process, the budget for the specific target to cover is computed as $\frac{SB_i}{n_i}$, where SB_i is the remaining budget and n_i is the estimated number of remaining targets to be covered. We estimate the number of targets to be covered by subtracting from the total number of branches of the Control-Flow Graph the number of branches already covered and/or used as targets (but not covered because they are infeasible or difficult to cover) at iteration i. Note that this is a conservative estimation, due to the possible collateral coverage of non target branches in the remaining iterations.

4 Empirical evaluation of OCELOT

The *goal* of the study is to compare the two test case generation methods implemented in OCELOT, *i.e.,* MOSA, a whole suite approach, and LIPS, an iterative single target approach. The *quality focus* of the study is the effectiveness and the efficiency of the two test case generation approaches, as well as the effort

#	Function name	Program name	LOC	Branches	Cyclomatic complexity
1	check_ISBN	bibclean	85	29	21
2	cliparc	spice	136	64	32
3	clip_line	spice	85	56	28
4	clip_to_circle	spice	117	44	22
5	Csqrt		26	6	3
6	gimp_cmyk_to_rgb	gimp	28	2	1
7	gimp_cmyk_to_rgb_int	gimp	23	2	1
8	gimp_hsl_to_rgb	gimp	31	4	2
9	gimp_hsl_to_rgb_int	gimp	34	4	2
10	gimp_hsl_value	gimp	22	10	5
11	gimp_hsl_value_int	gimp	22	10	5
12	gimp_hsv_to_rgb	gimp	69	11	8
13	gimp_rgb_to_cmyk	gimp	36	8	4
14	gimp_rgb_to_hsl	gimp	51	14	7
15	gimp_rgb_to_hsl_int	gimp	58	14	7
16	gimp_rgb_to_hsv4	gimp	62	18	9
17	gimp_rgb_to_hsv_int	gimp	59	16	8
18	gimp_rgb_to_hwb	gimp	32	2	1
19	gimp_rgb_to_l_int	gimp	19	2	1
20	gradient_calc_bilinear_factor	gimp	30	6	3
21	gradient_calc_conical_asym_factor	gimp	35	6	3
22	gradient_calc_conical_sym_factor	gimp	43	8	4
23	gradient_calc_linear_factor	gimp	30	8	4
24	gradient_calc_radial_factor	gimp	29	6	3
25	gradient_calc_spiral_factor	gimp	37	8	4
26	gradient_calc_square_factor	gimp	29	6	3
27	triangle		21	14	7
28	gsl_poly_complex_solve_cubic	GLS	113	20	11
29	gsl_poly_complex_solve_quadratic	GLS	77	12	7
30	gsl_poly_eval_derivs	GLS	41	10	6
31	gsl_poly_solve_cubic	GLS	73	14	8
32	gsl_poly_solve_quadratic	GLS	60	12	7
33	sglib_int_array_binary_search	SGLIB	32	8	5
34	sglib_int_array_heap_sort	SGLIB	80	28	15
35	sglib_int_array_quick_sort	SGLIB	102	30	16

Table 1: C functions used in the study

required for the definition of the oracle of the generated test cases. The *context* of the study consists of 35 open-source C functions, with a total of 605 branches, taken from different programs, in particular from `gimp`, an open source GNU image manipulation software, `GSL`, the GNU Scientific Library, `SGLIB`, a generic library for C, and `spice`, an analogue circuit simulator. We selected these functions since they have been used in previous work on test case generation for C language [20]. It is worth noting that, since the current implementation of OCELOT does not properly support the generation of test cases for functions having complex data types as input (e.g. pointers to struct), we selected only a subset of functions from the chosen programs. The main characteristics of the object programs are summarized in Table 1.

4.1 Research Questions and Analysis Method

The study is steered by the following research questions:

- **RQ$_1$** (**Effectiveness**): *Which is the coverage of MOSA as compared to LIPS when generating test cases for procedural code?*
- **RQ$_2$** (**Efficiency**): *Which is the execution time of MOSA as compared to LIPS when generating test cases for procedural code?*

– **RQ**$_3$ (**Oracle Cost**): *Which is the size of the test suite generated by MOSA as compared to the size of the test suite generated by LIPS?*

To address the three research questions we run the MOSA and LIPS 30 times for each object function and compute the average performance of the two approaches. Specifically:

– for **RQ**$_1$ we compare the average percentage of branches covered by each approach for each function.
– for **RQ**$_2$ we compare the average running time required by each approach for each function. The execution time was measured using a machine with Intel Core i7 processor running at 3.1 GHz with 4GB RAM.
– for **RQ**$_3$ we measure the average size of the test suite generated by each approach for each function.

We also statistically analyze the achieved results. Statistical significance is measured with the *Wilcoxon's test* [7], with a p-value threshold of 0.05. Significant p-values indicate that the corresponding null hypothesis can be rejected in favor of the alternative one, *i.e.,* one of the approaches reaches a higher coverage (**RQ**$_1$), it is faster in term of running time (**RQ**$_2$), or it generates smaller test suites (**RQ**$_3$). Other than testing the null hypothesis, we use the Vargha-Delaney ($\hat{A}_{12}$) statistical test [33] to measure the magnitude of difference between the results achieved by the two experimented approaches. Vargha-Delaney ($\hat{A}_{12}$) statistic also classifies the magnitude of the obtained effect size value into four different levels (*negligible*, *small*, *medium*, and *large*). It's important to note that in our experiments we setup the population size to 100 individuals and the search budget is 200.000 evaluations. Moreover the crossover probability is 0.90.

4.2 Analysis of the Results and Discussion

In this section we discuss the achieved results aiming at answering the research questions previously formulated. Table 2 shows the achieved results along with p-values obtained from Wilcoxon test [7]. The table also shows the effect size metric from Vargha-Delaney ($\hat{A}_{12}$) statistic [33], indicating also the magnitude of the difference.

RQ$_1$ **(Effectiveness)**. The first part of Table 2 summarizes the results in term of coverage achieved by MOSA and LIPS. The overall average coverage was 84.73% for MOSA and and 86.29% for LIPS. Also, LIPS is significantly better in *10 out of 35 cases* with an effect size `large` or `medium` in *8 cases*. Instead, MOSA achieves a significantly higher coverage just in two cases: once with a `small` effect size and once with a `large` effect size. Moreover, we can notice that when LIPS outperforms MOSA, the coverage increases between 0.28% and 15.83%; on the other hand, in the only case where MOSA performs better with a `large` effect size, the difference in terms of coverage is of 8.6% with respect to LIPS.

RQ$_2$ **(Efficiency)**. The second part of Table 2 shows the results achieved in terms of *efficiency*, measured as time spent for the generation of the test suites.

	Coverage					Execution time					Test suite size				
#	LIPS	MOSA	p-value	$\hat{A}_{12}$	Magnitude	LIPS	MOSA	p-value	$\hat{A}_{12}$	Magnitude	LIPS	MOSA	p-value	$\hat{A}_{12}$	Magnitude
1	86.21%	86.21%	1.000	0.50	negligible	**36.30**	59.10	**<0.001**	**1.00**	**large**	9.37	**7.43**	**<0.001**	**0.10**	**large**
2	94.95%	95.00%	0.276	0.47	negligible	**7.80**	26.10	**<0.001**	**1.00**	**large**	18.67	**15.67**	**<0.001**	**0.05**	**large**
3	**87.50%**	85.00%	**<0.001**	**1.00**	**large**	**15.00**	37.00	**<0.001**	**1.00**	**large**	10.90	9.40	0.022	0.24	large
4	86.59%	87.05%	0.659	0.55	negligible	**8.80**	31.40	**<0.001**	**1.00**	**large**	16.10	14.40	0.125	0.35	small
5	83.33%	83.33%	1.000	0.50	negligible	**6.87**	9.13	**<0.001**	**1.00**	**large**	3.47	**2.43**	**<0.001**	**0.12**	**large**
6	100.00%	100.00%	1.000	0.50	negligible	**0.00**	7.53	**<0.001**	**1.00**	**large**	4.00	**2.00**	**<0.001**	**0.00**	**large**
7	88.33%	80.00%	0.086	0.58	small	**1.53**	7.53	**<0.001**	**0.97**	**large**	2.77	**1.60**	**<0.001**	**0.07**	**large**
8	91.67%	92.50%	0.395	0.48	negligible	**2.43**	8.70	**<0.001**	**0.97**	**large**	4.33	**2.70**	**<0.001**	**0.12**	**large**
9	**93.33%**	86.67%	**0.019**	**0.63**	**small**	**1.93**	9.50	**<0.001**	**1.00**	**large**	4.47	**2.47**	**<0.001**	**0.06**	**large**
10	100.00%	100.00%	1.000	0.50	negligible	**0.00**	10.23	**<0.001**	**1.00**	**large**	6.67	**4.60**	**<0.001**	**0.00**	**large**
11	100.00%	100.00%	1.000	0.50	negligible	**0.00**	10.60	**<0.001**	**1.00**	**large**	6.37	**4.73**	**<0.001**	**0.05**	**large**
12	87.27%	**90.00%**	**0.004**	**0.35**	**small**	**4.93**	12.47	**<0.001**	**0.97**	**large**	10.20	**7.90**	**<0.001**	**0.00**	**large**
13	100.00%	100.00%	1.000	0.50	negligible	**0.00**	10.67	**<0.001**	**1.00**	**large**	4.90	**3.93**	**<0.001**	**0.14**	**large**
14	**85.00%**	78.57%	**<0.001**	**0.95**	**large**	**4.97**	11.83	**<0.001**	**1.00**	**large**	5.97	**3.60**	**<0.001**	**0.00**	**large**
15	**92.86%**	89.05%	**<0.001**	**0.77**	**large**	**7.07**	12.03	**<0.001**	**1.00**	**large**	5.87	**4.43**	**<0.001**	**0.09**	**large**
16	83.33%	83.33%	1.000	0.50	negligible	**7.53**	15.23	**<0.001**	**1.00**	**large**	5.30	**4.50**	**<0.001**	**0.17**	**large**
17	**86.67%**	83.12%	**<0.001**	**0.78**	**large**	**7.87**	15.83	**<0.001**	**1.00**	**large**	6.37	**5.20**	**<0.001**	**0.09**	**large**
18	51.67%	50.00%	0.167	0.52	negligible	**7.97**	9.10	**<0.001**	**0.98**	**large**	2.07	**1.00**	**<0.001**	**0.00**	**large**
19	100.00%	100.00%	1.000	0.50	negligible	**0.00**	8.03	**<0.001**	**1.00**	**large**	3.00	**2.00**	**<0.001**	**0.00**	**large**
20	84.44%	83.33%	0.157	0.53	negligible	**2.53**	8.53	**<0.001**	**1.00**	**large**	5.17	**3.00**	**<0.001**	**0.02**	**large**
21	83.33%	83.33%	1.000	0.50	negligible	**4.07**	11.07	**<0.001**	**1.00**	**large**	5.00	**3.00**	**<0.001**	**0.00**	**large**
22	86.67%	87.50%	0.080	0.47	negligible	**4.30**	11.67	**<0.001**	**1.00**	**large**	5.87	**4.00**	**<0.001**	**0.03**	**large**
23	87.92%	87.50%	0.285	0.52	negligible	**2.43**	9.03	**<0.001**	**1.00**	**large**	6.13	**4.00**	**<0.001**	**0.02**	**large**
24	**87.78%**	83.33%	**0.006**	**0.63**	**small**	**2.60**	8.23	**<0.001**	**1.00**	**large**	5.53	**3.00**	**<0.001**	**0.03**	**large**
25	87.08%	87.50%	0.167	0.48	negligible	**3.97**	11.07	**<0.001**	**1.00**	**large**	5.33	**3.50**	**<0.001**	**0.03**	**large**
26	**88.89%**	83.33%	**<0.001**	**0.67**	**medium**	**2.47**	8.07	**<0.001**	**1.00**	**large**	5.70	**3.00**	**<0.001**	**0.00**	**large**
27	88.89%	88.89%	1.000	0.50	negligible	**6.07**	12.10	**<0.001**	**1.00**	**large**	12.13	**7.80**	**<0.001**	**0.00**	**large**
28	**58.33%**	52.73%	**<0.001**	**0.79**	**large**	**4.10**	16.33	**<0.001**	**1.00**	**large**	5.87	**3.97**	**<0.001**	**0.03**	**large**
29	58.33%	58.33%	1.000	0.50	negligible	**3.50**	10.50	**<0.001**	**1.00**	**large**	4.00	**3.00**	**<0.001**	**0.00**	**large**
30	100.00%	100.00%	1.000	0.50	negligible	**0.00**	24.27	**<0.001**	**1.00**	**large**	2.40	**1.63**	**<0.001**	**0.19**	**large**
31	55.00%	**63.50%**	**<0.001**	**0.17**	**large**	**3.63**	14.60	**<0.001**	**1.00**	**large**	5.00	5.03	0.400	0.52	negligible
32	58.33%	58.61%	0.167	0.48	negligible	**3.23**	10.40	**<0.001**	**1.00**	**large**	4.00	**3.03**	**<0.001**	**0.02**	**large**
33	**100.00%**	84.17%	**<0.001**	**0.82**	**large**	**0.07**	11.57	**<0.001**	**1.00**	**large**	4.37	**1.43**	**<0.001**	**0.00**	**large**
34	100.00%	100.00%	1.000	0.50	negligible	**0.00**	18.63	**<0.001**	**1.00**	**large**	3.17	**2.20**	**<0.001**	**0.16**	**large**
35	**96.67%**	93.89%	**<0.001**	**0.83**	**large**	**12.17**	20.10	**<0.001**	**1.00**	**large**	4.63	**3.17**	**<0.001**	**0.18**	**large**

Table 2: Comparison of results achieved achieved by LIPS and MOSA.

Results are clearly in favor of LIPS. The overall average execution time for MOSA is 14.80 seconds, while LIPS spent, on average, 5.03 seconds for each function, with an improvement with respect to MOSA of about 66%. The improvement in terms of execution time is also supported by statistical tests. Specifically, the execution time of LIPS is statistically lower than the execution time of MOSA in *all the cases*, with a *large* effect size. It is worth noting that LIPS is faster even when it is able to achieve a significantly higher coverage. The most evident difference in terms of execution time can be observed in the case of function `gsl_poly_eval_derivs` (#30): MOSA spent about 24.27 seconds for the overall test suite generation process, while LIPS always needed less than a second. On this function the two approaches achieve exactly the same level of coverage (100%). In order to have more insights on why the iterative single target approach is faster than the whole test suite approach, we launched LIPS and MOSA on the function which requires the highest execution time for both the approaches (*i.e.,* `cliparc`) and used a Java profiler (VisualVM) to check at which step MOSA requires more time. We observed that the bottleneck in MOSA is represented by the algorithm used to compare the solutions, *i.e.,* ranking the solutions in different Pareto-fronts. LIPS does not need such an algorithm, thus saving execution time.

RQ_3 (Oracle cost). The third part of Table 2 shows the average size of the test suites generated by each approach for all the functions under test. The results show that, on average, MOSA generates about 4.4 test cases, compared to

	Coverage					Execution time					Test suite size				
#	**LIPS***	**MOSA**	**p-value**	**$\hat{A}_{12}$**	**Magnitude**	**LIPS***	**MOSA**	**p-value**	**$\hat{A}_{12}$**	**Magnitude**	**LIPS***	**MOSA**	**p-value**	**$\hat{A}_{12}$**	**Magnitude**
1	85.29%	85.75%	0.285	0.48	negligible	**33.37**	55.63	**<0.001**	**1.00**	**large**	**5.10**	7.30	**<0.001**	**0.94**	**large**
2	94.79%	95.05%	0.063	0.41	small	**4.57**	26.27	**<0.001**	**1.00**	**large**	**13.40**	15.73	**<0.001**	**0.94**	**large**
3	85.18%	85.54%	0.228	0.59	small	**14.20**	36.80	**<0.001**	**1.00**	**large**	9.40	8.50	0.100	0.67	medium
4	85.91%	85.91%	0.562	0.52	negligible	**6.30**	32.50	**<0.001**	**1.00**	**large**	12.40	12.70	0.408	0.54	negligible
5	83.33%	83.33%	1.000	0.50	negligible	**5.20**	8.07	**<0.001**	**1.00**	**large**	2.47	2.53	0.307	0.53	negligible
6	100.00%	100.00%	1.000	0.50	negligible	**0.00**	6.03	**<0.001**	**1.00**	**large**	2.00	2.00	1.000	0.50	negligible
7	86.67%	85.00%	0.391	0.52	negligible	**1.50**	6.53	**<0.001**	**0.95**	**large**	1.73	1.70	0.392	0.48	negligible
8	91.67%	94.17%	0.200	0.45	negligible	**1.67**	6.77	**<0.001**	**1.00**	**large**	2.67	2.77	0.200	0.55	negligible
9	90.00%	91.67%	0.301	0.47	negligible	**2.33**	7.13	**<0.001**	**1.00**	**large**	2.60	2.67	0.301	0.53	negligible
10	100.00%	100.00%	1.000	0.50	negligible	**0.00**	8.30	**<0.001**	**1.00**	**large**	**4.43**	4.70	**0.020**	**0.63**	**small**
11	100.00%	100.00%	1.000	0.50	negligible	**0.00**	8.43	**<0.001**	**1.00**	**large**	4.60	4.57	0.446	0.49	negligible
12	88.18%	89.39%	0.115	0.43	negligible	**3.43**	10.63	**<0.001**	**1.00**	**large**	7.70	7.83	0.115	0.57	negligible
13	100.00%	100.00%	1.000	0.50	negligible	**0.00**	8.20	**<0.001**	**1.00**	**large**	3.77	3.73	0.414	0.48	negligible
14	**83.81%**	78.57%	**<0.001**	**0.87**	**large**	**4.73**	11.07	**<0.001**	**1.00**	**large**	4.07	**3.67**	**0.012**	**0.35**	**small**
15	89.29%	89.76%	0.307	0.47	negligible	**0.90**	12.00	**<0.001**	**1.00**	**large**	4.57	4.63	0.247	0.55	negligible
16	83.33%	83.15%	0.167	0.52	negligible	**6.20**	14.07	**<0.001**	**1.00**	**large**	**4.20**	4.57	**0.002**	**0.68**	**medium**
17	84.79%	83.96%	0.155	0.57	negligible	**3.20**	12.10	**<0.001**	**1.00**	**large**	5.10	5.23	0.252	0.54	negligible
18	53.33%	50.00%	0.080	0.53	negligible	**5.23**	7.03	**<0.001**	**1.00**	**large**	1.07	1.00	0.080	0.47	negligible
19	100.00%	100.00%	1.000	0.50	negligible	**0.00**	5.53	**<0.001**	**1.00**	**large**	2.00	2.00	1.000	0.50	negligible
20	82.22%	82.78%	0.322	0.48	negligible	**2.50**	8.10	**<0.001**	**1.00**	**large**	2.93	2.97	0.322	0.52	negligible
21	81.67%	83.33%	0.041	0.45	negligible	**3.37**	8.13	**<0.001**	**1.00**	**large**	2.90	3.00	0.041	0.55	negligible
22	87.50%	87.50%	1.000	0.50	negligible	**3.00**	9.07	**<0.001**	**1.00**	**large**	4.00	4.00	1.000	0.50	negligible
23	88.33%	87.92%	0.285	0.52	negligible	**1.87**	8.93	**<0.001**	**1.00**	**large**	4.07	4.03	0.285	0.48	negligible
24	**90.56%**	83.33%	**<0.001**	**0.72**	**medium**	**1.53**	7.70	**<0.001**	**1.00**	**large**	3.43	**3.00**	**<0.001**	**0.28**	**medium**
25	87.50%	87.50%	1.000	0.50	negligible	**3.03**	9.10	**<0.001**	**1.00**	**large**	3.33	3.53	0.062	0.60	small
26	86.11%	83.33%	0.010	0.58	small	**2.03**	7.77	**<0.001**	**1.00**	**large**	3.17	**3.00**	**0.011**	**0.42**	**small**
27	87.59%	**88.89%**	**0.003**	**0.38**	**small**	**5.20**	12.03	**<0.001**	**1.00**	**large**	**7.43**	7.77	**0.013**	**0.64**	**small**
28	45.45%	**53.48%**	**<0.001**	**0.17**	**large**	**3.03**	15.80	**<0.001**	**1.00**	**large**	**3.00**	4.10	**<0.001**	**0.83**	**large**
29	58.33%	58.33%	0.167	0.50	negligible	**3.07**	10.03	**<0.001**	**1.00**	**large**	3.00	3.00	1.000	0.50	negligible
30	100.00%	100.00%	1.000	0.50	negligible	**0.00**	25.17	**<0.001**	**1.00**	**large**	1.63	1.63	0.578	0.51	negligible
31	55.00%	**58.50%**	**0.001**	**0.37**	**small**	**3.20**	14.20	**<0.001**	**1.00**	**large**	**4.00**	4.43	**0.001**	**0.63**	**small**
32	58.33%	58.61%	0.167	0.48	negligible	**3.10**	10.13	**<0.001**	**1.00**	**large**	3.00	3.03	0.167	0.52	negligible
33	**100.00%**	82.50%	**<0.001**	**0.85**	**large**	**0.43**	11.23	**<0.001**	**1.00**	**large**	2.50	**1.30**	**<0.001**	**0.09**	**large**
34	100.00%	100.00%	1.000	0.50	negligible	**0.00**	18.77	**<0.001**	**1.00**	**large**	2.77	**2.13**	**0.009**	**0.33**	**medium**
35	94.67%	94.11%	0.173	0.56	negligible	**11.63**	20.77	**<0.001**	**1.00**	**large**	4.27	**3.30**	**0.001**	**0.27**	**medium**

Table 3: Comparison of the results achieved by LIPS* and MOSA.

about 6.1 test cases of LIPS. This means that MOSA generates test suites that are 28.0% smaller, on average. The differences between the size of the generated test suites is significant with a `large` effect size in almost all the cases. It is worth noting that, in one of the 3 cases where the effect size is not large, MOSA achieves a significantly higher level of coverage as compared to LIPS.

The results achieved are quite expected since LIPS has been defined to efficiently use the search budget and maximize the coverage through the inclusion in the test suite of test cases covering branches not selected as target. Thus, it does not take into account the size of the test suite explicitly, as done, for instance, by the approach proposed by Harman *et al.* [15], but rather the underlying strategy often results in the inclusion of redundant test cases.

We also implemented a revised version of LIPS (indicated as LIPS*) where we avoid the inclusion in the test suite of test cases covering branches not selected

Test suite sizes after greedy minimization																
#	**LIPS**	**MOSA**	**p-value**	**$\hat{A}_{12}$**	**Magnitude**	#	**LIPS**	**MOSA**	**p-value**	**$\hat{A}_{12}$**	**Magnitude**	#	**LIPS**	**MOSA**	**p-value**	**$\hat{A}_{12}$**	**Magnitude**
1	4.00	4.03	0.434	0.51	negligible	13	3.13	3.00	0.120	0.44	negligible	25	3.07	3.17	0.158	0.55	negligible
2	11.00	10.87	0.273	0.46	negligible	14	4.00	**3.13**	**<0.001**	**0.11**	**large**	26	3.33	**3.00**	**<0.001**	**0.33**	**medium**
3	5.70	5.40	0.289	0.42	small	15	4.10	4.17	0.230	0.53	negligible	27	6.00	6.03	0.167	0.52	negligible
4	7.30	8.00	0.220	0.60	small	16	4.00	4.00	1.000	0.50	negligible	28	4.87	**3.97**	**<0.001**	**0.21**	**large**
5	2.13	2.13	0.505	0.50	negligible	17	4.00	4.00	1.000	0.50	negligible	29	3.00	3.00	1.000	0.50	negligible
6	2.00	2.00	1.000	0.50	negligible	18	1.03	1.00	0.167	0.48	negligible	30	1.17	1.27	0.178	0.55	negligible
7	1.77	1.60	0.086	0.42	small	19	2.00	2.00	1.000	0.50	negligible	31	**4.00**	5.03	**<0.001**	**0.83**	**large**
8	2.67	2.70	0.395	0.52	negligible	20	3.07	3.00	0.157	0.47	negligible	32	3.00	3.03	0.167	0.52	negligible
9	2.73	**2.47**	**0.019**	**0.37**	**small**	21	3.00	3.00	1.000	0.50	negligible	33	2.00	**1.37**	**<0.001**	**0.18**	**large**
10	4.23	4.20	0.382	0.48	negligible	22	3.93	4.00	0.080	0.53	negligible	34	1.90	1.87	0.457	0.49	negligible
11	4.20	4.20	0.504	0.50	negligible	23	4.03	4.00	0.285	0.48	negligible	35	1.90	1.87	0.457	0.49	negligible
12	**6.60**	6.90	**0.004**	**0.65**	**small**	24	3.27	**3.00**	**0.006**	**0.37**	**small**						

Table 4: Test suite sizes after greedy minimization for LIPS and MOSA.

as target. Table 3 shows the comparison between LIPS* and MOSA. The two approaches attain levels of coverage and test suite size very similar. They both achieve a significantly higher coverage in only 3 cases. About test suite size, MOSA generates smaller test suites in 6 cases, while LIPS* in 7 cases. It is worth noting that a difference in terms of coverage always implies a difference in terms of test suite size. Excluding such cases, only LIPS* achieves a *large* difference in terms of test suite size. Nevertheless, LIPS* still maintains a significantly higher efficiency in all the cases. This proves that LIPS privileges coverage as for test suite size. However, in our opinion this is not a limitation of the approach, as the size of the generated test suites can be easily reduced by using well-known minimization techniques [37].

To verify the effect of test suite minimization, Table 4 shows the comparison between the size of the test suites generated by MOSA and LIPS after minimizing the test suites using a greedy algorithm [37] [17]. It is worth noting that in order to have a fair comparison, the test suite minimization was applied also on the test suites generated by MOSA (even if minimization is implicit in MOSA). As we can see, the differences in terms of test suite size are radically ironed out after the minimization. As expected, the only significant differences concern some of the functions for which the achieved coverage is significantly different. In addition, the average time spent for the minimization task is always less than a second (nearly 0 seconds), hence its effect on the execution time is negligible.

4.3 Threats to Validity

This section discusses the threats to the validity of our empirical evaluation.

Construct Validity. We used three metrics widely adopted in literature: branch coverage, execution time and number of generated test cases [24]. In the context of our study, these metrics provides a good estimation of effectiveness (code coverage), efficiency (execution time) and oracle effort (test suite size). Another threat consists of the methodology used to compare LIPS with MOSA. Considering that an implementation of MOSA for the C language is not publicly available, we had to implement the approach in our tool. However, we strictly followed the definition of the algorithm provided by Panichella *et al.* [29]. Since also LIPS has been implemented in the same tool, the comparison of the two approach is much fairer, since it is not influenced by the underlying technology. Another threat to the construct validity consists of the meta-heuristic used in the study. Considering that MOSA is strictly based on GAs, due to its multi-objectives nature, we decided to limit our study to this kind of algorithm. However, in the future we plan to integrate other search-based algorithms in LIPS.

Internal Validity. We ran test data generation techniques 30 times for each subject program and reported average results together with statistical evidence to address the random nature of the GAs themselves [2]. The tuning of the GA's parameters is another factor that could affect the internal validity of this work. However, in the context of test data generation it is not easy to find good settings that significantly outperform the default values suggested in the literature [3]. For this reason, we used the default values widely used in literature.

External Validity. We considered 35 open-source functions taken from different programs. We selected these functions since they have been used in previous work on test case generation [20] for C language. Functions were selected to have small and quite large samples, as well as samples with low and high cyclomatic complexity. Also, `triangle` and `bibclean` are used in search-based software testing [16, 24], while `Csqrt` represents a valuable testing scenario since it contains a condition really hard to cover, namely a comparison between a double and a fixed number (0.0). Also, besides numerical input values, the considered functions take as input structures (`gimp_rgb_to_hsl`) and strings (`check_ISBN`) as well. However, in order to corroborate our findings, we plan to replicate the study on a wider range of programs. Our results are different than those achieved in the context of OO programming [13]. We cannot state if these differences are related to LIPS or concern the different characteristics between procedural and OO code. For this reason, in the future, we plan to implement LIPS in the context of Evosuite [12] and compare it with other whole suite approaches.

Conclusion Validity. We used appropriate statistical tests coupled with enough repetitions of the experiments. In particular, we used the Wilcoxon test [7] to test the significance of the differences and the Vargha-Delaney statistic [33] to estimate the magnitude and the effect size of the observed differences.

5 Conclusion and Future Work

We presented OCELOT, a tool for test case generation for C programs. OCELOT implements the most efficient and effective state-of-the-art whole suite approach MOSA [29], adapted for procedural test case representation, and a new iterative single-target approach named LIPS (Linearly Independent Path based Search), inspired by the baseline method for the construction of a maximal set of linearly independent paths [34] and designed to avoid search budget wasting. An empirical study conducted on 35 C functions was carried out to compare the two test data generations approaches implemented in OCELOT. The results indicate that the iterative approach provides a better or comparable level of coverage with respect to whole suite approach with a much lower execution time. The main weakness of LIPS with respect to MOSA is represented by the size of the generated test suites. However, after applying a test suite minimization approach based on greedy algorithm [17] the difference in terms of test suite size between the two techniques becomes negligible, without affecting execution time.

As future work, we plan to replicate the study on a larger dataset of programs and also in the context of OO programs. We also plan to implement in OCELOT different search algorithms and compare them with genetic algorithms.

References

1. Arcuri, A.: It does matter how you normalise the branch distance in search based software testing. In: International Conference on Software Testing, Verification and Validation. pp. 205–214. IEEE (2010)

2. Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: International Conference on Software Engineering. pp. 1–10. IEEE (2011)
3. Arcuri, A., Fraser, G.: Parameter tuning or default values? an empirical investigation in search-based software engineering. Empirical Software Engineering 18(3), 594–623 (2013)
4. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. IEEE Transactions on Software Engineering 41(5), 507–525 (2015)
5. Beizer, B.: Software testing techniques. Van Nostrand Reinhold Co. (1990)
6. Chang, K.H., CROSS II, J.H., Carlisle, W.H., Liao, S.S.: A performance evaluation of heuristics-based test case generation methods for software branch coverage. International Journal of Software Engineering and Knowledge Engineering 6(04), 585–608 (1996)
7. Conover, W.J.: Practical nonparametric statistics (1980)
8. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: Nsga-ii. IEEE Transactions on Evolutionary Computation 6(2), 182–197 (2002)
9. Durillo, J.J., Nebro, A.J.: jmetal: A java framework for multi-objective optimization. Advances in Engineering Software 42(10), 760–771 (2011)
10. Ferguson, R., Korel, B.: The chaining approach for software test data generation. ACM Transactions on Software Engineering and Methodology 5(1), 63–86 (1996)
11. Ferrer, J., Chicano, F., Alba, E.: Evolutionary algorithms for the multi-objective test data generation problem. Software: Practice and Experience 42(11), 1331–1362 (2012)
12. Fraser, G., Arcuri, A.: Evosuite: automatic test suite generation for object-oriented software. In: Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. pp. 416–419. ACM (2011)
13. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Transactions on Software Engineering 39(2), 276–291 (2013)
14. Gallagher, M.J., Narasimhan, V.L.: Adtest: A test data generation suite for ada software systems. IEEE Transactions on Software Engineering 23(8), 473–484 (1997)
15. Harman, M., Kim, S.G., Lakhotia, K., McMinn, P., Yoo, S.: Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In: International Conference on Software Testing, Verification, and Validation Workshops. pp. 182–191. IEEE (2010)
16. Harman, M., McMinn, P.: A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In: International Symposium on Software Testing and Analysis. pp. 73–83. ACM (2007)
17. Harrold, M.J., Gupta, R., Soffa, M.L.: A methodology for controlling the size of a test suite. ACM Transactions on Software Engineering and Methodology 2(3), 270–285 (1993)
18. Jones, B.F., Sthamer, H.H., Eyres, D.E.: Automatic structural testing using genetic algorithms. Software Engineering Journal 11(5), 299–306 (1996)
19. Korel, B.: Automated software test data generation. IEEE Transactions on Software Engineering 16(8), 870–879 (1990)
20. Lakhotia, K., Harman, M., Gross, H.: Austin: An open source tool for search based software testing of c programs. Information and Software Technology 55(1), 112–125 (2013)

21. Larson, E., Austin, T.: High coverage detection of input-related security faults. Ann Arbor 1001(48105), 29 (2003)
22. Laumanns, M., Thiele, L., Deb, K., Zitzler, E.: Combining convergence and diversity in evolutionary multiobjective optimization. Evolutionary computation 10(3), 263–282 (2002)
23. McCabe, T.J.: A complexity measure. IEEE Transactions on Software Engineering (4), 308–320 (1976)
24. McMinn, P.: Search-based software test data generation: a survey. Software Testing, Verification and Reliability 14(2), 105–156 (2004)
25. McMinn, P.: IGUANA: Input Generation Using Automated Novel Algorithms. a plug and play research tool. Tech. Rep. CS-07-14, Department of Computer Science, University of Sheffield (2007)
26. Michael, C.C., McGraw, G., Schatz, M.A.: Generating software test data by evolution. IEEE Transactions on Software Engineering 27(12), 1085–1110 (2001)
27. Michael, C.C., McGraw, G.E., Schatz, M.A., Walton, C.C.: Genetic algorithms for dynamic test data generation. In: International Conference Automated Software Engineering. pp. 307–308. IEEE (1997)
28. Oster, N., Saglietti, F.: Automatic test data generation by multi-objective optimisation. In: SAFECOMP. vol. 4166, pp. 426–438. Springer (2006)
29. Panichella, A., Kifetew, F.M., Tonella, P.: Reformulating branch coverage as a many-objective optimization problem. In: International Conference on Software Testing, Verification and Validation. pp. 1–10. IEEE (2015)
30. Pargas, R.P., Harrold, M.J., Peck, R.R.: Test-data generation using genetic algorithms. Software Testing, Verification and Reliability 9(4), 263–282 (1999)
31. Tonella, P.: Evolutionary testing of classes. In: ACM SIGSOFT Software Engineering Notes. vol. 29, pp. 119–128. ACM (2004)
32. Tracey, N., Clark, J., Mander, K., McDermid, J.: An automated framework for structural test-data generation. In: International Conference on Automated Software Engineering. pp. 285–288. IEEE (1998)
33. Vargha, A., Delaney, H.D.: A critique and improvement of the cl common language effect size statistics of mcgraw and wong. Journal of Educational and Behavioral Statistics 25(2), 101–132 (2000)
34. Watson, A.H., McCabe, T.J., Wallace, D.R.: Structured testing: A testing methodology using the cyclomatic complexity metric. NIST special Publication 500(235), 1–114 (1996)
35. Wegener, J., Baresel, A., Sthamer, H.: Evolutionary test environment for automatic structural testing. Information and Software Technology 43(14), 841–854 (2001)
36. Xanthakis, S., Ellis, C., Skourlas, C., Le Gall, A., Katsikas, S., Karapoulios, K.: Application of genetic algorithms to software testing. In: International Conference on Software Engineering and its Applications. pp. 625–636 (1992)
37. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability 22(2), 67–120 (2012)
38. Zitzler, E., Künzli, S.: Indicator-based selection in multiobjective search. In: Parallel Problem Solving from Nature-PPSN VIII. pp. 832–842. Springer (2004)
39. Zitzler, E., Laumanns, M., Thiele, L.: Spea2: Improving the strength pareto evolutionary algorithm. Tech. rep. (2001)